1. Property based testing (QuickTest)

- The inner workings of QuickTest do not really matter. The important thing is coming up and writting properties to verify correctness.
- Remember to explicitly write the types of the predicates so that QuickCheck can provide appropriate examples.

2. Lazy evaluation

2.1. WHNF (= Weak Head Normal Form)

An expression is in WHNF if any of the following are true:

1. The expression is a **constructor**;
   - If the expression is a constructor that is being pattern matched on (e.g. a constructor inside a `case ... of`, then it will be reduced depending on the value of the constructor.
2. The expression is an **anonymous function**, i.e. a lambda expression;
   - If lambda is being applied to all of its arguments, then it is reduced depending on their value.
3. The expression is a **function applied to too few arguments**.

3. Higher-kinded abstractions

3.1. Functor

```
class Functor m where
  fmap :: (a -> b) -> f a -> f b
```

- Generalisation of `map` for arbitrary containers
- **Examples:**
  - `[]` (List) where `fmap = map`,
  - `Maybe` where `fmap  mapMaybe`, but rather `fmap f Nothing = Nothing, fmap  f (Just x) = f x`, and
  - `IO` where `fmap` comes from the `IO` Monad.
- There is an alias for `fmap` namely `<$>`, e.g. `map (+1) [1..3] == (+1) <$>  [1..3]`.

3.1.1. Functor laws

```
fmap id = id`
fmap (f . g) = fmap f . fmap g`
```

3.2. Aplicative

```
class Functor f => Applicative f where
  pure :: a -> f a
  <*>  :: f (a -> b) -> f a -> f b
```

- Sometimes called *Applicative functor*
- Found in the `Control.Applicative` module.
- **Examples:**
  - `Maybe`
  - `List`
  - `IO`
- `liftA2 :: (Applicative f) => (a -> b -> c) -> f a -> f b -> f c` takes a binary function an *lifts* it to a function that operates on two functors.

3.3. Monoid

```
class Monoid a where
  mempty  :: a
  mappend :: a -> a -> a
  mconcat :: [a] -> a
  mconcat = foldr mconcat mempty
```

- We often write `mappend` using infix notation with `<>`.

3.3.1. Monoid laws

```
-- mempty is the identity element for <>
mempty <> x = x
x <> mempty = x

-- The <> operator is left and right associative
(x <> y) <> z = x <> (y <> z)
```

3.4. Monad

```
class Functor m => Monad m where
  (>>=)  :: m a -> (a -> m b) -> m b
  return :: a -> m a
```

3.4.1. Do notation

- Remember that each do block maps to a monad, and some monads don't do what we intuitively think they do. Consider

  ```
  a <- do f <- [1, 2]
          s <- ['a', 'b']
          return (f, s)

  a == [(1, 'a'), (1, 'b'), (2, 'a'), (2, 'b')]
  ```

  where the list monad comes into place (think of **list comprehensions** in this case).

3.4.2. Aplicative

3.4.3. Traversable